TP 4 – Communication inter-processus

Vous devez faire tous les exercices.

Exercice 1 - `kill`, `signal`

Cet exercice porte sur les programmes `sigsend` et `sigcatch` dont les sources sont disponibles dans `/pub/FISE_OSSE11/syscall/sigsend.c` et `/pub/FISE_OSSE11/syscall/sigcatch.c`.

1. Complétez `sigsend.c` et compilez le.

```
sh$ gcc -o sigsend sigsend.c
```

2. Décrivez le comportement du programme `sigcatch` :

   main

   lignes 27 à 32 Attache le ..... à tous les signaux de ... à ... .

   lignes 29 à 30 Affiche un message d'erreur pour les signaux qui ne sont pas ..... .

   lignes 36 à 39 Boucle infinie : affichage d'un ..... puis attente d'un ..... .

   callback (gestionnaire de signal)

   lignes 15 à 19 Affiche le .....

3. a. Compilez le.

```
sh$ gcc -o sigcatch sigcatch.c
```

   b. Lancez deux fois le programme `sigcatch`, dans deux terminaux différents.

```
sh$ xterm -e ./sigcatch &
sh$ xterm -e ./sigcatch &
```

   c. Dans quel état sont les deux processus `sigcatch` ?

4. Envoyez quelques signaux à chacun des deux processus.

```
sh$ ./sigsend hup <pid1>
sh$ ./sigsend 10 <pid2>
```

5. a. Tapez <CTRL-C> dans les terminaux des deux processus `sigcatch`.

   b. Se sont-ils terminés ?

   c. Que se passe-t-il quand <CTRL-C> est tapé dans un terminal ?

6. a. Tapez <CTRL-Z> dans les terminaux des processus `sigcatch`.

   b. Se sont-ils mis en pause ?

   c. Que se passe-t-il quand <CTRL-Z> est tapé dans un terminal ?

7. a. Envoyez le signal `SIGSTOP` à un des processus `sigcatch`.

```
sh$ ./sigsend STOP <pid1>
```

   b. A-t-il été reçu par le processus ?

   c. Envoyez lui maintenant le signal `SIGUSR1`. L'a-t-il reçu ?

   d. Tapez <CTRL-C> dans le terminal du processus. A-t-il reçu le signal `SIGINT` ?

   e. Envoyez lui maintenant le signal `SIGCONT`. L'a-t-il reçu ?

   f. Qu'en est il des signaux `SIGUSR1` et `SIGINT` précédents ?

8. Recommencez l'expérimentation précédente en envoyant plusieurs fois le signal `SIGUSR1`.

   Combien de réceptions du signal `SIGUSR1` sont traitées ?

9. Terminez les deux processus en utilisant la commande shell `kill` (qui est une version améliorée du programme `sigsend`).

Exercice 2 - flux, fichiers réguliers et fifo

Cet exercice porte sur le programme `mycat`, dont les sources sont disponibles dans le fichier `/pub/FISE_OSSE11/syscall/mycat.c`.

Ce programme met en évidence le comportement des appels système

```
ssize_t read(int fd, void* buf, size_t count)
```

et

```
ssize_t write(int fd, const void* buf, size_t count)
```

en fonction du type de fichier.

La table ci-dessous résume (en supposant que les arguments sont valides) le comportement attendu :

	mode	bloquant	retour		SIGPIPE
			0	-1	
read	regulier	jamais	fin fichier	non	jamais
write	regulier	jamais	jamais	full	jamais
read	fifo	V & $ne \neq 0$	V & $ne = 0$	jamais	jamais
write	fifo	P & $nl \neq 0$	jamais	spa & $nl = 0$	$nl = 0$

ne = nombre d'écrivains — V = FIFO vide

nl = nombre de lecteurs — P = FIFO pleine

$full$ = disque plein ou dépassement de quota

spa = signal `SIGPIPE` attrapé ou ignoré

1. Décrivez le comportement du programme `mycat` :

   lignes 24 + 13 à 19
   Attache le ..... au signal `SIGPIPE`. Le gestionnaire affiche un message, attend ... secondes, puis termine le processus avec 0 comme code de retour.

   lignes 26 à 38 boucle infinie :

   ligne 28 Lecture d'un caractère `c` du flux ..... ;

   ligne 29 à 31 Si une ..... de lecture s'est produite, on écrit un message d'erreur et on ..... le processus ;

   ligne 32 à 34 Si la .... du flux d'entrée est détectée, on écrit un ..... sur le flux ....., puis on ..... 0.1 seconde ;

   ligne 36 Dans les autres cas, on écrit le caractère `c` sur le flux ..... .

2. Compilez le fichier `/pub/FISE_OSSE11/syscall/mycat.c`.

```
sh$ gcc -Wall -Wextra -o mycat /pub/FISE_OSSE11/syscall/mycat.c
```

3. Expérimentation de `mycat` sur des flux réguliers.

   a. Dans quatre terminaux différents, lancez deux écrivains sur le fichier `file` et deux lecteurs sur ce même fichier.

```
sh$ xterm -e bash -c "./mycat > file" &
sh$ xterm -e bash -c "./mycat > file" &
sh$ xterm -e bash -c "./mycat < file" &
sh$ xterm -e bash -c "./mycat < file" &
```

b. Tapez les entrées suivantes dans les fenêtres des écrivains, en regardant leurs résultats dans les fenêtres des lecteurs.

aaaa<ENTER>bbbb<ENTER>	dans la fenêtre du premier écrivain.
cccc<ENTER>dddd<ENTER>eeee<ENTER>	dans la fenêtre du deuxième écrivain.
ffff<ENTER>gggg<ENTER>	dans la fenêtre du premier écrivain.

c. Stoppez les processus lecteurs avec <CTRL-S> (vous pourrez les réveiller avec <CTRL-Q>).

d. Soit P_1 et P_2 deux processus, et f fichier régulier :

— Une lecture sur f n'est pas bloquante. Si on est en fin de fichier, le noyau renvoie une valeur indiquant la fin de fichier.

— Si P_1 et P_2 écrivent dans f à la même position, la donnée de f à cette position sera celle de la ..... écriture.

— Si P_1 et P_2 lisent dans f à la même position, les données lues seront les mêmes si il n'y a pas eu ..... dans f à cette position entre les .....

— Si P_1 fait une lecture et P_2 une écriture dans f à la position ℓ, P_1 reçoit la donnée écrite par P_2 si :

i. la lecture de P_1 est faite ..... l'écriture de P_2,

ii. il n'y a pas eu d'autre ..... entre la lecture de P_1 et l'écriture de P_2.

e. Terminez les 4 processus.

4. Expérimentation de `mycat` sur des flux FIFO.

a. Créez un fichier `fifo`, correspondant à un pipe nommé.

```
sh$ mkfifo fifo
```

b. Dans quatre terminaux différents, lancez deux écrivains et deux lecteurs sur le fichier `fifo`.

```
sh$ xterm -e bash -c "./mycat > fifo" &
sh$ xterm -e bash -c "./mycat > fifo" &
sh$ xterm -e bash -c "./mycat < fifo" &
sh$ xterm -e bash -c "./mycat < fifo" &
```

c. Tapez les entrées suivantes dans les fenêtres des écrivains, en regardant leurs résultats dans les fenêtres des lecteurs.

aaaa<ENTER>bbbb<ENTER>	dans la fenêtre du premier écrivain.
cccc<ENTER>dddd<ENTER>	dans la fenêtre du deuxième écrivain.
eeee<ENTER>ffff<ENTER>	dans la fenêtre du premier écrivain.

d. Tuez les processus écrivains.

e. Relancez un nouveau écrivain et tapez gggg<ENTER> dans son terminal.

f. Tuez les processus lecteurs.

g. Relancez un nouveau écrivain et tapez `hhhh`<ENTER> dans son terminal.

h. Soit P_1 et P_2 deux processus et f un fichier FIFO :

— Une lecture sur f est bloquante, si f est ..... et qu'il existe encore un .....

— Une écriture sur f est bloquante, si f est pleine et qu'il existe encore un .....

— En absence de lecteur, une écriture sur une f génère l'émission du signal ..... Si ce signal n'est ni attrapé, ni ignoré, ceci entraine ..... du processus. Si ce signal est attrapé ou ignoré, l'écriture renvoie une ..... et la variable ..... reçoit la valeur `EPIPE`.

— Si P_1 et P_2 écrivent dans f, les données seront écrites soit l'une derrière l'autre, soit enchevêtrées.

— Si P_1 et P_2 lisent dans f, ils ne peuvent pas lire la ..... donnée.

— Si P_1 fait une lecture et P_2 une écriture dans f, P_1 reçoit la donnée écrite par P_2 si :

i. la FIFO f est .....,

ii. il n'y a pas d'autre ..... entre la lecture de P_1 et l'écriture de P_2,

iii. il n'y a pas d'autre ..... entre la lecture de P_1 et l'écriture de P_2.

Si ces conditions sont réalisées, l'ordre chronologique de la lecture de P_1 et l'écriture de $P2$ .....

Exercice 3 - `pipe`, `fcntl`

L'appel système

```
int pipe(int fd[2])
```

permet de créer un fichier FIFO (aussi appelé *pipe*) non nommé.

En cas de succès, le descripteur `fd[0]` permet de lire le contenu du pipe, et le descripteur `fd[1]` permet d'écrire dans le pipe.

Le but de cet exercice est de déterminer la taille (en octets) d'un pipe non nommé.

1. Une approche naïve mais facile à mettre en place est d'écrire un code C qui :
   a. crée un pipe non nommé ;
   b. boucle indéfiniment, en écrivant à chaque étape un caractère dans le pipe ;
   c. affiche à l'écran, avant chaque écriture dans le pipe, le nombre de caractères précédemment écrits dans le pipe.

   Il suffit alors de compiler puis lancer le programme, et d'attendre que l'écriture devienne bloquante (ce qui arrivera lorsque le pipe sera plein).

   Implantez et testez cette approche.

2. Consultez le man de la fonction `fcntl` pour trouver une meilleure approche. Testez votre nouvelle approche et comparez avec le résultat obtenu à la question précédente.

3. Si vous avez le temps, adaptez votre code pour déterminer la taille (en octets) d'un pipe nommé.

   note : Vous devrez faire appel aux fonctions `mkfifo` et `unlink`.

Exercice 4 - `pause`, `alarm`

L'appel système

```
int pause()
```

met le processus courant en pause, et ce jusqu'à réception d'un signal.

L'appel système

```
int alarm(int nb)
```

indique au système d'envoyer un signal `SIGALRM` au processus courant dans `nb` secondes.

1. Copiez le fichier `/pub/FISE_OSSE11/syscall/mysleep.c`, et écrivez le code de la fonction `mysleep`. Cette fonction prend en argument un entier positif `nb`, ne renvoie rien, et a pour effet de suspendre l'exécution du processus courant pendant `nb` secondes.

2. Compilez votre code et testez le.

   note : Votre programme doit s'arrêter au bout de `nb` secondes, et afficher le message `done`. Si ce n'est pas le cas, vous avez probablement un problème dans le traitement des signaux.